Polyhedral Transformations of Explicitly Parallel Programs

Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar
Department of Computer Science, Rice University
{prasanth,shirako,vsarkar}@rice.edu

ABSTRACT

The polyhedral model is a powerful algebraic framework that has enabled significant advances to analyses and transformations of sequential affine (sub)programs, relative to traditional AST-based approaches. However, given the rapid growth of parallel software, there is a need for increased experiences with using polyhedral frameworks for analysis and transformations of explicitly parallel programs. An interesting side effect of supporting explicitly parallel programs is that doing so can also enable analysis and transformation of programs with unanalyzable data accesses within a polyhedral framework, since explicit parallelism can often mitigate the imprecision that accompanies unanalyzable data accesses arising from a variety of sources, including unrestricted pointer aliasing, unknown function calls, and certain classes of non-affine constructs. In this paper, we address the problem of extending polyhedral frameworks to enable analysis and transformation of programs that contain both explicit parallelism and unanalyzable data accesses.

A summary of our approach is as follows. As in past work, we first enable conservative dependence analysis of a given region of code; for simplicity, we use an approach based on *dummy variables* that can work with any polyhedral tool that supports *access functions*. After obtaining conservative dependences, we use the Fourier-Motzkin elimination method to remove all dummy variables. Next, we identify *happens-before* relations from the explicitly parallel constructs, and subtract their complement from the conservative dependences. The resulting set of dependences can then be passed on to a polyhedral transformation tool, such as PLuTo, to enable transformation of explicitly-parallel programs with unanalyzable data accesses.

To motivate our approach, we studied 18 explicitly-parallel OpenMP benchmarks from the Rodinia benchmark suite, and found that these benchmarks use six classes of non-affine constructs that are commonly found in parallel scientific applications: 1) Non-affine subscript expressions, 2) Indirect array subscripts, 3) Use of `structs`, 4) Calls to user-defined functions, 5) Non-affine loop bounds, and 6) Non-affine `if` conditions. While there are known techniques from past work to enable automatic analysis for some of these non-affine constructs in polyhedral frameworks, we show that the use of explicit parallelism can enable a larger set of polyhedral transformations for some of these programs, compared to what might have been possible if the input program was sequential.

Keywords

Explicit parallelism, Polyhedral transformations

1. INTRODUCTION

A key challenge for optimizing compilers is to keep up with the increasing complexity related to locality and parallelism in modern computers, especially as computer vendors head towards new designs for extreme-scale processors and exascale systems. Classical AST-based optimizers typically focus on one particular objective at a time, such as vectorization, locality or parallelism. On the other hand, polyhedral transformation frameworks are able to support arbitrarily complex sequences of transformations of perfectly/imperfectly nested loops in a unified framework. The benefit of this unified formulation can be seen in polyhedral optimizers such as PLuTo [6, 7], which has even been extended and specialized to integrate SIMD constraints [17]. Polyhedral frameworks achieve this generality in transformation by restricting the class of programs that are supported to those that do not have "unanalyzable" data or control flow. In the original formulation of polyhedral frameworks, all array subscripts, loop bounds, and `if` conditions in "analyzable" programs were required to be affine functions of loop index variables and global parameters. However, decades of research since then has led to a great expansion of programs that can be considered analyzable by polyhedral frameworks. The main remaining constraints include restrictions on pointer usage in order to eliminate aliasing, on recursion, and on unstructured control flow [3]. While recent work [23] has shown how to combine polyhedral and AST-based techniques, it would be interesting to try and further generalize the polyhedral model so that it can deal with both explicit parallelism and unanalyzable constructs.

This work is motivated by the observation that software with explicit parallelism is on the rise, and that explicit parallelism can be used to enable larger sets of polyhedral transformations (by mitigating conservative dependences), compared to what might have been possible if the input program was sequential. Our work focuses on explicitly-parallel programs that specify potential parallelism, rather than actual

IMPACT 2015
Fifth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2015, Amsterdam, The Netherlands
In conjunction with HiPEAC 2015.

http://impact.gforge.inria.fr/impact2015

parallelism. Thus, explicit parallelism is simply a specification of a partial order, which is traditionally referred to as a *happens-before* relation [29]. Hence, we can mitigate conservative dependences arising from unanalyzable constructs by subtracting the complement of the happens-before relation from the conservative dependences, since dependences can only occur among statement instances that are ordered by the *happens-before* relation. In this paper, we will restrict our attention to explicitly-parallel programs that satisfy the *serial elision* property i.e., the property that removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics.

A summary of our approach is as follows. As in past work, we first enable conservative dependence analysis of a given region of code; for simplicity, we use an approach based on *dummy variables* that can work with any polyhedral tool that supports *access functions*. After obtaining conservative dependences, we use the Fourier-Motzkin elimination method to remove all dummy variables. Next, we identify *happens-before* relations from the explicitly parallel constructs, and subtract their complement from the conservative dependences. The resulting set of dependences can then be passed on to a polyhedral transformation tool, such as PLuTo, to enable transformation of explicitly-parallel programs with unanalyzable data accesses.

To motivate our approach, we studied 18 explicitly-parallel OpenMP benchmarks from the Rodinia benchmark suite, and found that these benchmarks use six classes of non-affine constructs that are commonly found in parallel scientific applications: 1) Non-affine subscript expressions, 2) Indirect array subscripts, 3) Use of `structs`, 4) Calls to user-defined functions, 5) Non-affine loop bounds, and 6) Non-affine `if` conditions. While there are known techniques from past work to enable automatic analysis for some of these non-affine constructs in polyhedral frameworks, we show that the use of explicit parallelism can enable a larger set of polyhedral transformations for some of these programs (due to conservative dependences), compared to what might have been possible if the input program was sequential. Recent work on the PENCIL intermediate language [3], has also shown similar results through the use of directives/pragmas that can be generated from higher-level domain-specific languages (DSLs). A key difference between our approach and the PENCIL approach is that we are interested in leveraging happens-before information from programs written in general-purpose explicitly parallel languages, such as OpenMP and X10, whereas PENCIL is focused on supporting DSLs in which certain coding rules are enforced related to pointer aliasing, recursion, unstructured control flow, etc.

The rest of the paper is organized as follows. Section 2 introduces the parallel constructs considered in this paper. Section 3 motivates the problem and provides an overview of our proposed approach. Section 4 discuss some potential limitations of existing polyhedral frameworks. Section 5 discusses details of our approach to enable polyhedral transformations of explicitly parallel programs. Section 6 presents a case study to illustrate the potential of our proposed approach. Section 7 and Section 8 summarize related work and our conclusions.

2. EXPLICITLY-PARALLEL PROGRAMS

The biggest difference between sequential programs and explicitly-parallel programs is that sequential programs specify a total execution order, whereas the execution of an explicitly-parallel program can be viewed as a partial order, which is traditionally referred to as a *happens-before* relation. Thus, we can mitigate conservative dependences arising from non-affine constructs by subtracting the complement of the happens-before relation from the conservative dependences, since dependences can only occur among statement instances that are ordered by the *happens-before* relation [29].

In this paper, we will restrict our attention to explicitly-parallel programs that satisfy the *serial elision* property i.e., the property that removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics. As a first step, we will focus on two kinds of loop-level parallel constructs, *doall* and *doacross*, both of which satisfy the serial elision property. We briefly summarize these constructs in the context of OpenMP [20], which is a widely used parallel programming model. The OpenMP standard already supports doall parallelism as in the form of parallel loops. In OpenMP 4.1, doacross parallelism is expected to be supported as extensions to the `ordered` construct [24, 21].

2.1 Doall parallelism

The OpenMP loop construct, "`#pragma omp for`", is specified immediately before a `for` loop and indicates that the iterations of the loop have no happens-before dependence and hence can be executed in parallel. A barrier, i.e., an all-to-all synchronization point, is implied immediately after the parallel loop construct. The implicit barrier may be omitted if a `nowait` clause is specified on the loop construct.

The `reduction(`*op*`:` *list*`)` clause, which is attached to a `for` loop construct, indicates that the parallel loop contains a reduction computation whose operator is specified by *op* (e.g., `+` or `max`) and the reduction variables are specified in *list*. Note that both scalar and array reductions are supported in OpenMP-FORTRAN while only scalar reductions are supported in OpenMP-C. For convenience, we assume the availability of array reductions in OpenMP-C (as proposed in [13]), when discussing examples in this paper. The aggregation for reduction is handled by the OpenMP runtime and the target variables specified in *list* have no other happens-before dependences from the compiler viewpoint.

The example in Section 3.2 shows an example usage of the OpenMP `for` loop construct, where the loops in lines 2, 8, 20 and 24 of Figure 3 are annotated as doall loops.

2.2 Doacross parallelism

Doacross parallelism [10] is supported as a proposed extension to the OpenMP ordered construct. The `ordered(`*n*`)` clause, which is attached to `for` loop construct, indicates the availability of doacross parallelism in a set of *n* perfectly nested loops, "`#pragma omp for ordered(`*n*`)`". The *n* loops form an *n*-dimensional iteration space in which an iteration instance can only depend on lexicographically earlier iterations (thereby satisfying the serial elision property). This pragma specifies the rank/dimensionality for the iteration vectors specified in the `depend source/sink` clauses introduced below.

In order to specify cross-iteration dependences within the

n-dimensional space, the ordered construct is extended with `depend(`*type*`: vec)` clauses. Here, *type* is `source` or `sink` and *vec* is an n-dimensional vector whose elements are simple expressions of the form, $vec = (x_1 \pm c_1, x_2 \pm c_2, \cdots, x_n \pm c_n)$, where x_k is the k-th loop index and c_k is an integer constant ($1 \leq k \leq n$). The `ordered depend(sink:` *vec*`)` [`depend(sink:` *vec*`)` [...]] construct specifies a dependence sink and indicates a synchronization point that waits for the iterations specified by *vec* to reach the dependence sources. Note that *vec* for `sink` must be lexicographically smaller than the current iteration vector - i.e., $(x_1, x_2, \cdots, x_n)$. At runtime, a `depend(sink:` *vec*`)` clause becomes a no-op if its *vec* indicates a point outside the iteration space. The `ordered depend(source:` *vec*`)` construct specifies a dependence source and indicates the point at which dependences on the current iteration $vec = (x_1, x_2, \cdots, x_n)$ are satisfied .

The example in Section 3.1 shows the usage of a doacross construct. In Figure 1, the `ordered(3)` clause at line 2 specifies a 3-level doacross loop nest. Within the specified triply nested loops, the ordered `depend` construct in lines 6-10 indicates a dependence sink (i.e., synchronization point) that depends on nine dependence sources in iterations `(t,i-1,j-1)`, `(t,i-1,j)`, `(t,i-1,j+1)`, `(t,i,j-1)`, `(t-1,i,j+1)`, `(t-1,i+1,j-1)`, `(t-1,i+1,j)`, and `(t-1,i+1,j+1)` while the `ordered depend(source: t,i,j)` construct at line 16 specifies the location of the dependence source arising from current iteration `(t,i,j)`.

3. MOTIVATING EXAMPLES

To motivate the proposed approach, we discuss two explicitly parallel kernels with data accesses that may be considered unanalyzable by some polyhedral frameworks. The first example uses C nested arrays, which may have unrestricted pointer aliasing in general. The second example uses non-affine linearized array subscripts, that would require a delinearization analysis to make them analyzable by polyhedral frameworks.

3.1 2-D Gauss Seidel Computation

The first example is a 2-dimensional 9 point Gauss Seidel computation. In Figure 1, the statement `S` is enclosed

```
1  // Assume array A is a nested array
2  #pragma omp parallel for ordered(3)
3  for (t = 0; t <= _PB_TSTEPS - 1; t++) {
4    for (i = 1; i<= _PB_N - 2; i++) {
5      for (j = 1; j <= _PB_N - 2; j++) {
6  #pragma omp ordered depend(sink: t,i-1,j-1)\
7  depend(sink: t,i-1,j) depend(sink: t,i-1,j+1)\
8  depend(sink: t,i,j-1) depend(sink: t-1,i,j+1)\
9  depend(sink: t-1,i+1,j-1) depend(sink: t-1,\
10 i+1,j) depend(sink: t-1,i+1,j+1)
11       A[i][j] = (A[i-1][j-1] + A[i-1][j] +
12                  A[i-1][j+1] + A[i][j-1] +
13                  A[i][j] + A[i][j+1] +
14                  A[i+1][j-1] + A[i+1][j] +
15                  A[i+1][j+1])/9.0; //S
16 #pragma omp ordered depend(source: t,i,j)
17     }
18   }
19 }
```

Figure 1: Input 2-D Gauss Seidel computation

in triply nested loops that are defined as a doacross nest where the ordered and depend clauses are new extensions proposed for OpenMP 4.1 [24, 21]. Even though the loop nest has affine accesses on a single array, C's unrestricted aliasing semantics for nested arrays can prevent a sound compiler analysis from detecting the exact cross-iteration dependences. However, the happens-before relations through explicit doacross parallelism can provide sufficient dependence information to enable loop skewing and tiling transformations to be performed so as to improve both locality and parallelism granularity, as shown in Figure 2.

```
1  // Assume array A is a nested array
2  #pragma omp parallel for ordered(3)
3  for (c1 = ...) {
4   for (c3 = ...) {
5    for (c5 = ...) {
6  #pragma omp ordered depend(sink: c1-1, c3, c5)\
7                      depend(sink: c1, c3-1, c5)\
8                      depend(sink: c1, c3, c5-1)
9     for (c7 = ...) {
10     for (c9 = ...) {
11      for (c11 = ...) {
12       A[c9-c7][c11-c7-c9] =
13   (A[c9-c7-1][c11-c7-c9-1] + A[c9-c7-1][c11-c7-c9]
14   +A[c9-c7-1][c11-c7-c9+1] + A[c9-c7][c11-c7-c9-1]
15   +A[c9-c7][c11-c7-c9] + A[c9-c7][c11-c7-c9+1]
16   +A[c9-c7+1][c11-c7-c9-1] + A[c9-c7+1][c11-c7-c9]
17   +A[c9-c7+1][c11-c7-c9+1]) / 9.0; //S
18     }}}
19 #pragma omp ordered depend (source: c1, c3, c5)
20 }}}
```

Figure 2: Transformed 2-D Gauss Seidel computation

To illustrate the potential performance impact of these transformations, we timed three versions of this Gauss-Seidel computation when executed for 100 time steps on a 2000×2000 matrix. These timings were obtained on a quad eight-core 3.86GHz IBM Power7 system (32 cores total). The timings were as follows, and clearly illustrate the potential benefit of the transformations in Figure 2:

- Original parallel version (Figure 1) on 32 cores: 7.39 seconds
- Sequential version: 3.93 seconds
- Optimized parallel version (Figure 2) on 32 cores: 0.32 seconds

3.2 Particle Filter

The second example is *particle filter* from Rodinia benchmarks [8] and is shown in Figure 3. The second loop nest in this kernel contains non-affine array subscripts `ind[x*countOnes+y]` and indirect array subscripts `I[ind[x*countOnes+y]]`. We observe that these non-affine accesses do not pose an obstacle to recent polyhedral tools, since existing delinearization techniques [16] can be used to handle the `ind[x*countOnes+y]` case, and the fact that the array `I` is read-only in the kernel can be used to handle the `I[ind[x*countOnes+y]]` case.. However, we would still like to discuss how the use of the parallel constructs can enable analysis and transformations, even in the absence of techniques such as delinearization.

Figure 4 shows the transformed version of this kernel after loop fusion is performed. The legality of loop fusion is easily established by the fact that all variables that cross multiple loops have affine accesses with no fusion-preventing dependences. The key information needed from the parallel program is that the second loop (lines 7-17 in Figure 3) has no loop carried dependence. This ensures that the fused loop can also be made parallel.

```
#pragma omp parallel for
for(x = 0; x < Nparticles; x++){
  arrayX[x] += 1 + 5*randn(seed, x);
  arrayY[x] += -2 + 2*randn(seed, x);
}

#pragma omp parallel for private(y, indX, indY)
for(x = 0; x < Nparticles; x++){
  for(y = 0; y < countOnes; y++){
    indX = roundDouble(arrayX[x]) + objxy[y*2+1];
    indY = roundDouble(arrayY[x]) + objxy[y*2];
    ind[x*countOnes+y] = fabs(indX ... indY ...);
    ...
    likelihood[x] += ...I[ind[x*countOnes+y]]...
  }
  ...
}

#pragma omp parallel for
for(x = 0; x < Nparticles; x++){
  weights[x] = weights[x] * exp(likelihood[x]);

#pragma omp parallel for private(x) \
  reduction(+:sumWeights)
for(x = 0; x < Nparticles; x++)
  sumWeights += weights[x];
}
```

Figure 3: *Particle filter* kernel having Non-affine and Indirect array subscripts

```
#pragma omp parallel for private(x, y, \
  indX, indY) reduction(+:sumWeights)
for(x = 0; x < Nparticles; x++){
  arrayX[x] += 1 + 5*randn(seed, x);
  arrayY[x] += -2 + 2*randn(seed, x);
  for(y = 0; y < countOnes; y++){
    indX = roundDouble(arrayX[x]) + objxy[y*2+1];
    indY = roundDouble(arrayY[x]) + objxy[y*2];
    ind[x*countOnes+y] = fabs(indX ... indY ...);
    ...
    likelihood[x] += ...I[ind[x*countOnes+y]]...
  }
  ...
  weights[x] = weights[x] * exp(likelihood[x]);
  sumWeights += weights[x];
}
```

Figure 4: *Particle filter* kernel after Loop fusion

3.3 Summary of our Approach

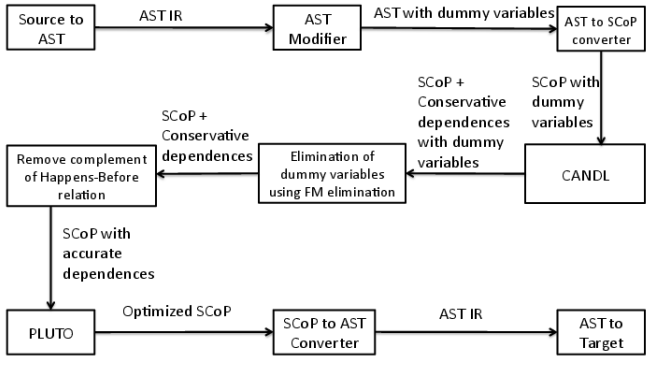

Figure 5: Overview of our approach

Our overall approach is summarized in Figure 5, which is being implemented as an extension to the PACE Compiler framework [1, 23], and consists of the following components: 1) Conversion from source code to AST (with support for doall and doacross parallel constructs), 2) AST Modifier (insertion of dummy variables), 3) AST to SCoP converter (with dummy variables), 4) Use of CANDL for conservative dependence analysis, 5) Use of Fourier-Motzkin elimination method to remove all dummy variables, 6) Identification of *happens-before* relations from the explicitly parallel constructs, and subtract their complement from the conservative dependences, 7) Communication of the resulting set of dependences to a polyhedral transformation tool, such as PLuTo, and 8) Generation of transformed AST from optimized SCoP. Of these, we have currently implemented steps 1) to 6), and are currently working on integrating the output from 6) with PLuTo, so as to complete steps 7) and 8).

4. POLYHEDRAL MODEL AND ITS LIMITATIONS

In general, our interest is in using the polyhedral model as an intermediate representation for performing compiler transformations for improved performance. Polyhedral frameworks transform selected regions in the input program into *Static Control Part* (SCoP) [12, 14], and capture precise dependence information among statement instances in the form of a dependence polyhedron over the iterators and global parameters. If there are any unanalyzable constructs in the SCoP, then there may be obstacles in constructing the dependence polyhedron. The good news is that decades of research has led to a great expansion of programs that can be considered analyzable by polyhedral frameworks, thereby reducing the impact of these limitations. The main remaining constraints include restrictions on pointer usage in order to eliminate aliasing, on recursion, and on unstructured control flow.

Based on a study of the Rodinia benchmarks [8], we have identified six common patterns in scientific applications that may be considered unanalyzable by some polyhedral frameworks: 1) Non-affine subscript expressions, 2) Indirect array subscripts, 3) Use of `structs`, 4) Calls to user-defined functions, 5) Non-affine loop bounds, and 6) Non-affine `if` conditions. Doerfert et.al. [11], performed a similar investigation recently on the applicability of Polly [15], a polyhedral framework for LLVM, on the SPEC 2000 benchmark suite and classified the root causes that prevented the application of polyhedral frameworks to these programs. We discuss two of the identified six common patterns (non-affine and indirect array subscripts) in Section 4.2 and Section 4.3. A more challenging limitation (unrestricted pointer aliasing) was already discussed in Section 3.1.

4.1 Background on polyhedral model

The polyhedral model is a flexible representation for perfect and imperfect loop nests with static predictable control. Loop nests amenable to this algebraic representation are called *Static Control parts* (SCoPs), roughly defined as a set of consecutive statements such that loop bounds, conditionals and array accesses are affine functions over iterators and global parameters invariant to the SCoP.

Iteration domain, $\mathcal{D}_S$: A statement $\mathcal{S}$ enclosed by 'm' loops is represented as m-dimensional polytope, referred to as an iteration domain [4].

Access relation, $\mathcal{A}$ maps from statement instances to the array elements accessed by those statement instances [28]. The access relation can be an access function, in which case the mapping is described using affine constraints over loop iterators and global parameters. In our conservative approach using *dummy variables*, the access relation, for an array access with non-affine subscripts, represents a read or write access to the entire array.

Dependence Polyhedra, $\mathcal{P}^{\mathcal{S}\rightarrow\mathcal{T}}$: captures all possible dependences between statements $\mathcal{S}$ and $\mathcal{T}$. Two instances $\vec{X_s}$ and $\vec{X_t}$, which belong to the iteration domains of statements $\mathcal{S}$ and $\mathcal{T}$ respectively, are said to be in dependence if they access the same array location and at least one of them is a write. Multiple dependence polyhedra may be required to capture all dependent instances between two statements, at least one for each pair of array references accessing the same array cell (scalars being a particular case of array). In the remainder of this section, we briefly summarize our approach to handling non-affine and indirect array subscripts in a polyhedral framework.

4.2 Non-affine Subscripts

Consider the kernel from Figure 6, part of the *hotspot* program from the Rodinia benchmark suite [8]. In this program,

```
1 int result[N], temp[N];
2 for (r = 0; r < row; r++)
3     for (c = 0; c < col; c++)
4         result[r*col+c]=temp[r*col+c]; // S
```

Figure 6: Part of *hotspot* kernel

`r*col+c` is a non-affine subscript expression used to index into the `result` array. As mentioned earlier, delinearization techniques can be used to make this example analyzable. However, our approach makes this example conservatively analyzable by introducing a *dummy variable* to capture the `r*col+c` value, and later uses explicit parallelism to mitigate the conservative dependence analysis.

4.3 Indirect Array Subscripts

Now, consider the sparse matrix-vector multiplication kernel in Figure 7. The statement $\mathcal{S}$ performs a non-affine read

```
1 for(i = 0; i < n; i++)
2   for(j = index[i]; j < index[i+1]; j++)
3         y[i] += A[j]*x[col[j]]; //S
```

Figure 7: Sparse matrix-vector multiplication

operation (`col[j]`) on array `x`. This particular access cannot be represented as an affine combination of iterators (`i`, `j`) and global variables (`n`). As we will see later in Section 5, our approach makes these indirect references analyzable by introducing a *dummy variable* to represent the subscript `col[j]` of array `x`, and considers `col[j]` itself as a read array reference of array `col`.

5. APPROACH DETAILS

This section presents the details of the proposed workflow, which was summarized in Section 3. For simplicity, our approach to handle conservative dependences is based on *dummy variables* that can work with any polyhedral tool that supports *access functions*; alternative approaches would have been possible if we had used access relations instead.

5.1 Terminology: Dummy vector

A *dummy vector* consists of *dummy variables* to represent non-affine subscript expressions and indirect array subscripts in the statement. These parameters are different from *iterators* and *parameters* in the polyhedral model. Each non-affine expression and indirect array subscript in a statement is uniquely associated with an element from the dummy vector corresponding to that statement. Now, each dynamic instance of a statement S is uniquely identified by its iteration vector ($\vec{i_S}$), dummy vector ($\vec{d_S}$) and parameter vector ($\vec{p}$). The kernel in Figure 8 contains two indirect array subscripts (`x[i][j], y[i][j]`) in the statement `S`. These subscripts are treated as *dummy variables* for that statement, and are replaced by the dummy variables, dmy_1 and dmy_2, respectively for subsequent computation of dependences in Figure 12.

5.2 Overall Algorithm

Algorithm 1 Overall algorithm for transformation of explicit parallel program with non-affine constructs

1: **Input:** SCoP
2: Let $\mathcal{P}$ be the set of dependence polyhedra computed over SCoP using conservative analysis with *dummy variables*.
3: Let $\mathcal{P}'$ be the set of dependence polyhedra obtained after elimination of the *dummy variables* from $\mathcal{P}$ using FM elimination.
4: Let $\mathcal{P}''$ be the set of dependence polyhedra after reflection of happens-before relations from explicitly parallel constructs (C) in $\mathcal{P}'$.
5: Forward SCoP and $\mathcal{P}''$ to a polyhedral optimizer, such as PLuTo [2].
6: **Output:** SCoP with optimized schedules

Algorithm 1 shows the overall approach at a high level, to handle non-affine constructs (step 2) and to use explicit parallelism to improve the accuracy of conservative dependences (step 4). In step 2, statements with non-affine expressions and indirect array subscripts are handled by the conservative approach with *dummy variables* while using exact dependence analysis techniques for regular statements. Finally, the resulting dependence polyhedra are passed to polyhedral optimizers to leverage existing loop transformations(step 5).

5.3 Conservative Approach

In this subsection, we summarize our conservative approach to compute dependence polyhedra for a SCoP with both regular statements, and statements with non-affine expressions or indirect array subscripts. In conservative approaches, the basic assumption for a compiler is that all memory accesses of an array in a statement can potentially conflict with other memory accesses of that array, or perhaps even memory accesses in other arrays (in the case of unrestricted aliasing). We use *dummy variables* instead of access relations for simplicity, since we use the *Scoplib* format when using CANDL.[1]

In case of non-affine expressions, we replace these expressions with *dummy variables* as part of pre-processing before

[1]We recently learned that CANDL also supports the *OpenScop* format which does supports access relations.

```
int A[N][N], x[N][N], y[N][N];
#pragma omp parallel for
for (i = 0; i < N; i++)
#pragma omp parallel for ordered(1)
    for (j = 0; j < N; c++)
    #pragma omp ordered depend(sink: j-1)
        A[j][i] = A[x[i][j]][y[i][j]]; // S
    #pragma omp ordered depend(source: j)
```

Figure 8: Input kernel

```
int A[N][N], x[N][N], y[N][N];
#pragma omp parallel for ordered(2)
for (j = 0; j < N; i++)
    for (i = 0; i < N; c++)
    #pragma omp ordered depend(sink: j-1, i)
        A[j][i] = A[x[i][j]][y[i][j]]; // S
    #pragma omp ordered depend(source: j, i)
```

Figure 9: Transformed kernel for better spatial locality

$$\mathcal{D}^S : \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \\ dmy_1 \\ dmy_2 \\ N \\ 1 \end{pmatrix} \geq 0$$

Figure 10: $\mathcal{D}^S$: Domain of statement S

$$\mathcal{D}'^S : \begin{pmatrix} 1 & 0 & 0 & 0 \\ -1 & 0 & 1 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq 0$$

Figure 11: $\mathcal{D}'^S$: Domain of statement S after elimination

Conservative Dependences ($\mathcal{P}$)	Dependences after elimination ($\mathcal{P}'$)	Happens-before relations (C)	Reflection ($\mathcal{P}'' = \mathcal{P}' \cap C$)
$\mathcal{P}_1^{S\to S}$: $i \leq i^1 - 1$ $j = dmy_1, i = dmy_2$ $0 \leq i, j \leq (N-1)$ $0 \leq i^1, j^1 \leq (N-1)$ $0 \leq dmy_1, dmy_2 \leq (N-1)$ $0 \leq dmy_1^1, dmy_2^1 \leq (N-1)$	$\mathcal{P}'^{S\to S}_1$: $i \leq i^1 - 1$ $0 \leq i, j \leq (N-1)$ $0 \leq i^1, j^1 \leq (N-1)$	$C_1^{S\to S}$: $i = i^1$	$\mathcal{P}''^{S\to S}_1 : \phi$
$\mathcal{P}_2^{S\to S}$: $i = i^1, j \leq j^1 - 1$ $j = dmy_1, i = dmy_2$ $0 \leq i, j \leq (N-1)$ $0 \leq i^1, j^1 \leq (N-1)$ $0 \leq dmy_1, dmy_2 \leq (N-1)$ $0 \leq dmy_1^1, dmy_2^1 \leq (N-1)$	$\mathcal{P}'^{S\to S}_2$: $i = i^1, j \leq j^1 - 1$ $0 \leq i, j \leq (N-1)$ $0 \leq i^1, j^1 \leq (N-1)$	$C_2^{S\to S}$: $i = i^1, j = j^1 - 1$	$\mathcal{P}''^{S\to S}_2$: $i = i^1, j = j^1 - 1$ $0 \leq i, j \leq (N-1)$ $0 \leq i^1, j^1 \leq (N-1)$

Figure 12: Dependences ($\mathcal{P}$) from conservative approach, Dependences ($\mathcal{P}'$) after elimination of additional parameters, Happens-before dependences from explicitly parallel constructs (C) and Dependences ($\mathcal{P}''$) after reflection of C to $\mathcal{P}'$

SCoP extraction. Then, we create affine inequalities from access ranges of these arrays, and eliminate these *dummy variables* from the SCoP using the FM elimination after the computation of conservative dependences. In case of indirect array subscripts, we associate the index arrays into the read arrays list of that statement after the computation of affine inequalities from access range of arrays. Algorithm 2 summarizes the steps involved in computing dependences in our conservative approach. For the kernel in Figure 8, the indirect array subscripts in statement `S` are replaced by *dummy variables* dmy_1, dmy_2 as part of pre-processing before SCoP extraction. Then, the affine inequalities for the *dummy variables* are created based on access range of array `A` and they are incorporated into iteration domain of the statement `S`, shown in Figure 10. After the computation of conservative dependences, the *dummy variables* are eliminated from the iteration domain using the FM elimination method, shown in Figure 11. First two columns of Figure 12 respectively show the conservative dependences with *dummy variables* and dependences after elimination of *dummy variables* for kernel in Figure 8.

Algorithm 2 Conservative Approach

1: **Input:** SCoP
2: **for** each statement S in SCoP **do**
3: **for** each array a in statement S that has non-affine expression/ indirect accesses in its subscripts **do**
4: **for** each subscript i in a that has non-affine expression/ indirect accesses **do**
5: Incorporate the affine inequalities, based on *access range* of array a for subscript i, into iteration domain of statement S
6: **if** subscript i has indirect array subscripts **then**
7: Associate index arrays into *read arrays* list of statement S
8: **end if**
9: **end for**
10: **end for**
11: **end for**
12: Forward SCoP to dependence analyzer
13: **Output:** Set of dependence polyhedra ($\mathcal{P}$)

5.4 Reflection of happens-before relations in dependence polyhedra

The set of dependence polyhedra in the source program is represented as $\mathcal{P} = \{\mathcal{P}_d^{S_i \to S_j} |$ there exists a dependence between source statement S_i and target statement S_j at depth $d\}$. Here depth represents the loop nest level that carries the data dependence. The doall and doacross parallel constructs are tied to specific loops and impose constraints on the dependence polyhedra whose source statement S_i and target statement S_j are enclosed in the specified loop and depth d is matched with the specified loop's nest level. As with dependence polyhedra, let $C_d^{S_i \to S_j}$ denote a constraint on possible dependences between source S_i and target S_j at depth $= d$, which is imposed by doall/doacross constructs and $x_k^{S_i}/x_k^{S_j}$ denote the k-th loop indexes of S_i/S_j respectively. A doall construct specified at depth d always imposes a constraint $C_d^{S_i \to S_j}$ such that $x_k^{S_i} = x_k^{S_j}$ for $1 \le k \le d$. On the other hand, a doacross construct with `ordered(n)` - i.e., n-level doacross parallelism - at depth d may impose multiple constraints $C_d^{S_i \to S_j}$, $C_{d+1}^{S_i \to S_j}$, ..., $C_{d+n-1}^{S_i \to S_j}$, which correspond to the `depend(sink: vec)` clauses within the loop body. Let `depend(sink:` $(x_d^{S_j}, \cdots, x_{m-1}^{S_j}, x_m^{S_j} - c_m, \cdots, x_{d+n-1}^{S_j} - c_{d+n-1}))$ denote a `depend` clause that specifies a cross-iteration dependence at depth $m \ge d$. The corresponding constraint is $C_m^{S_i \to S_j}$ such that $x_k^{S_i} = x_k^{S_j}$ for $1 \le k \le m-1$ & $x_k^{S_i} = x_k^{S_j} - c_k$ for $m \le k \le d+n-1$. For instance, Figure 8 has two constraints $C_1^{S \to S}$, which is imposed by the doall construct at the outermost level, and $C_2^{S \to S}$, which is imposed by the doacross construct at the second level, as shown in Figure 12.

Algorithm 3 shows how to compute the set of dependence polyhedra $\mathcal{P}''$ after reflecting the constraints due to the above explicit parallel constructs (i.e., set of constraints C), where the input $\mathcal{P}'$ is the set of conservative dependence polyhedra computed in Section 5.3. For each polyhedron $\mathcal{P}_d'^{S_i \to S_j}$ in $\mathcal{P}'$ (line 2), it searches for the matched constraint $C_d^{S_i \to S_j}$ in C (lines 3–9). If such a $C_d^{S_i \to S_j}$ is found, the modified dependence polyhedron after reflection is computed as: $\mathcal{P}_d''^{S_i \to S_j} = \mathcal{P}_d'^{S_i \to S_j} \cap C_d^{S_i \to S_j}$ (line 7). Otherwise, the dependence polyhedron $\mathcal{P}_d'^{S_i \to S_j}$ is left unchanged (line 11). The Reflection column of Figure 12 shows the dependence polyhedra for the kernel in Figure 8 after reflection. After the reflection of explicit parallelism onto dependences, the loop interchange transformation is performed for better data-locality and the transformed program is shown in Figure 9.

6. CASE STUDY

For all 18 benchmarks in the Rodinia suite, Table 1 summarizes 1) constructs used in the benchmarks that limit the use of some polyhedral frameworks, and 2) potential opportunities for polyhedral loop transformations that can be enabled by our proposed approach based on exploiting explicit parallelism. The constructs in 1) include non-affine array subscripts (NAS), indirect array accesses (I), use of structs (S), and use of function calls (F). For instance, the hotspot benchmark has non-affine array subscripts (NAS), but can benefit from loop fusion, skewing, tiling, and doacross transformations (by leveraging explicit parallelism to identify happens-before dependences). The table also shows that non-affine subscripts and function calls are common in this benchmark suite, while indirect array accesses and structs are found in a few benchmarks. In the remainder of this section, we illustrate an optimizing transformations on one of the Rodinia

Algorithm 3 Reflection of happens-before relations from explicitly parallel constructs

1: **Input:** conservative dependences $\mathcal{P}'$ and constraints C
2: **for** each dependence polyhedron $\mathcal{P}_d'^{S_i \to S_j}$ in $\mathcal{P}'$ **do**
3: found := false;
4: **for** each constraint $C_e^{S_k \to S_l}$ in C **do**
5: **if** $S_i = S_k$ & $S_j = S_l$ & $d = e$ **then**
6: found := true;
7: $\mathcal{P}_d''^{S_i \to S_j} = \mathcal{P}_d'^{S_i \to S_j} \cap C_e^{S_k \to S_l}$;
8: **end if**
9: **end for**
10: **if** found = false **then**
11: $\mathcal{P}_d''^{S_i \to S_j} = \mathcal{P}_d'^{S_i \to S_j}$;
12: **end if**
13: Add the reflected polyhedron $\mathcal{P}_d''^{S_i \to S_j}$ to $\mathcal{P}''$;
14: **end for**
15: **Output:** dependence polyhedra after reflection $\mathcal{P}''$

benchmarks, LU Decomposition. Another Rodinia benchmark, Particle Filter, was discussed earlier in Section 3.2.

The kernel of LU Decomposition (LUD) calculates solutions to a set of linear equations and Figure 13 shows a part of LUD kernel. In Figure 13, the `j` and `k` loops are parallel and that the k loop is parallel with a reduction on array `a` (using an extension to OpenMP to specify array reductions in C [13, 23]). In the access pattern $k*size+j$ of array a in

```
1  int a[size*size];
2  for (i=0; i <size; i++) {
3  #pragma omp parallel for
4    for (j=i; j <size; j++) {
5  #pragma omp parallel for reduction(+:a)
6      for (k=0; k<i; k++) {
7        a[i*size+j]-=a[i*size+k]*a[k*size+j]; //S1
8      }
9    }
10   ....
11 }
```

Figure 13: Input LUD Kernel

statement S1, each iteration of loop k results in accessing an element distant from *size* elements from current location. As a result, it exhibits poor spatial locality. In this scenario, the interchange of loops j, k preserves semantics and improves spatial locality. The code after performing loop permutation on the input kernel is shown in Figure 14.

```
1  int a[size*size];
2  for (i=0; i <size; i++) {
3  #pragma omp parallel for reduction(+:a) \
4                 private(j)
5    for (k=0; k<i; k++) {
6      for (j=i; j <size; j++) {
7        a[i*size+j]-=a[i*size+k]*a[k*size+j]; //S1
8      }
9    }
10     ....
11 }
```

Figure 14: LUD Kernel after permutation

7. RELATED WORK

In this section, we discuss related approaches in applying polyhedral transformations to programs with non-affine static parts, as well as related work on polyhedral analysis

Limitations

Kernel	NAS	I	S	F	Transformations
b+ tree		✓	✓		
backprop				✓	perm, fuse, vect
bfs		✓	✓		
cfd	✓			✓	
heartwall				✓	
hotspot	✓				doacross, fuse, skew, tile, vect
kmean				✓	perm, fuse, vect
lavaMD	✓	✓	✓		
leukocyte				✓	fuse, vect
lud	✓				perm, vect
mummergpu			✓	✓	
myocyte	✓			✓	
nn	✓			✓	
nw	✓			✓	doacross, skew, perm
particle filter	✓			✓	fuse, vect
path finder					doacross, skew, tile
srad	✓				
streamcluster	✓	✓	✓	✓	
Total	10	4	5	11	

Table 1: Limitations and possible transformations in Rodinia benchmarks (NAS: non-affine array subscript, I: indirect array access, S: structure, F: function, and perm/fuse/skew/tile/doacross/vect: loop permutation/fusion/skewing/tiling/doacross parallelism/vectorization)

of parallel programs. These approaches are broadly classified into compile-time approaches, run-time approaches and a combination of run-time and compile-time techniques.

Since non-affine static parts in programs are not directly representable in the polyhedral model, the compile-time approaches use conservative dependence analysis techniques to perform legal transformations on these programs. In general, conservative approaches are over-approximation techniques, and consider a large superset of existing dependences. As a result, only a subset of legal transformations can be applied, and profitable legal transformations that may yield better performance may be bypassed. There has been a significant effort to handle certain subsets of non-affine accesses, including polynomial accesses [19] in the polyhedral model, and indirect array subscripts [18] for array dependence analysis and loop transformations.

There has also been a large effort devoted to extending the polyhedral model to support non-affine extensions. The approach in [22] introduced techniques to perform dependence analysis in the case of nonlinear expressions in array subscripts and loop bounds. In this approach, the nonlinear constraints are not omitted, as in the generic conservative approach. It uses uninterpreted function symbols to represent non-linear expressions, and the proposed dependence analysis technique generates dependence relations by approximating with affine dependence relations, where as in our approach, conservative dependences are pruned after reflection of happens-before relations from explicit parallel constructs.

Run-time approaches such as the inspector/ executor strategy [5, 27, 26] have also gained significant attention to determine data reordering and better communication schedules at run-time. In these approaches, the inspector code, generated from the program, is executed at run-time and gathers information about non-affine static parts, such as index expressions. Then, the executor part of strategy performs optimizations based on the run-time information from inspector code. Recently, this inspector/ executor transformation strategy was integrated into the polyhedral framework [27] and combined with regular loop transformations to optimize programs with non-affine static parts. But, this work was restricted to only indirect array subscripts accessing read-only data, which is only one source of non-affine computations.

There have been other run-time approaches to handle real world programs with the polyhedral representation. The approach in [11] has shown that most of the issues stem from overly conservative approximation of dependences through static analyses of real world programs. The approach in [11] proposed a speculative polyhedral optimization techniques, a variant of just-in-time polyhedral optimization, and focused on two specific sources of false dependences (possible aliasing and non-affine subscript expressions). The proposed approach handles them by tuning the region of code not amenable to the polyhedral optimizations with the run-time information and speculatively specialized functions. In the process of auto tuning, it infers the common values of parameters, and generates the optimized variants of those regions of codes along with original region of code. These optimized variants of code will get executed when enabled by run-time parameters. Our approach differs from and is complementary to the approach in [11] in two ways: 1) We don't rely on auto tuning capabilities to improve the precision of dependences over non-linear array subscript expressions but we rely on explicit parallel directives to improve the precision of dependences instead, and 2) Our approach is completely static where the other approach has to keep several variants of original code for different values of parameters for run-time execution of program.

Recent work [25], has also shown the applicability of polyhedral optimizations using POLLY (an extension of the LLVM compiler) on over 50 real-world programs from different do-

mains. They have proposed extensions to POLLY to recognize and model multi-dimensional arrays using the polyhedral model, instead of the default approach of modeling them as indirect pointer accesses. In addition, the authors have extended the polyhedral modeling capability in POLLY to another generalized polyhedral model using semi-algebraic sets and real algebra [25]. This formalism is proposed to deal with polynomials in array subscript expressions along with affine expressions. Quantifier elimination has been used as a tool to remove the quantifiers generated while computing data dependences among array accesses involving polynomial expressions over loop iterators and parameters. Since the authors handle polynomial expressions in array subscripts, a sufficient condition was proposed to reduce dependence analysis problem for polynomial array subscript expressions to affine conditions. Our approach differs from their approach [25] in two ways: 1) The implementation of our approach is in a high-level AST, in which multi-dimensional array accesses are explicit and need not be reconstructed via delinearization (as in LLVM), and 2) Our approach does not have the worst case doubly exponential complexity of modeling polyhedra over semi-algebraic sets, as in [25], and can use explicit parallelism to handle more non-affine cases (such as indirect accesses) than their approach.

There has been a recent work on PENCIL [3], a platform-neutral compute intermediate language, aimed at facilitating automatic parallelization and optimization for execution on multi-threaded SIMD hardware for multiple high performance domain specific languages (DSLs). The language provides extensions and directives that allow users to supply information about dependences and memory access patterns to help the optimizer to perform optimizations better than in case of conservative optimizations. A key difference between our approach and the PENCIL approach is that we are interested in leveraging happens-before information from programs written in general-purpose explicitly parallel languages, such as OpenMP and X10, whereas PENCIL is focused on supporting DSLs in which certain coding rules are enforced related to pointer aliasing, recursion, unstructured control flow, etc.

A number of papers addressed the problem of data-flow analysis of explicitly parallel programs, including extensions of array data-flow analysis to data-parallel and/or task-parallel programs [9], and adaptation of array data-flow analysis to the X10 programs with finish/async parallelism [29]. In these approaches, the happens-before relations are first analyzed and the data-flow is computed based on the partial order imposed by happen-before relations. On the other hand, our approach first overestimates dependences based on the sequential order and subtracts the complement of the happen-before relations from the conservative dependences. While the work in [29] identifies potential data races, our approach does not treat potential data races as errors or dependences. Further, the main focus of our work is on transformations of explicitly parallel programs for improved performance, whereas the work in [9] and [29] is only focused on analysis.

8. CONCLUSIONS

This work is motivated by the observation that software with explicit parallelism is on the rise, and that explicit parallelism can be used to enable larger sets of polyhedral transformations (by mitigating conservative dependences), compared to what might have been possible if the input program was sequential. We introduced an approach that reflects happens-before constraints from explicitly parallel constructs in the dependence polyhedra to help mitigate conservative dependence analysis. In our approach, we subtract the complement of the happens-before relation from the conservative dependences, since dependences can only occur among statement instances that are ordered by the *happens-before* relation. The updated set of dependence can then be passed on to a polyhedral transformation tool, such as PLuTo, to enable transformation of explicitly parallel programs. Our approach to modeling non-affine constructs is based on access functions, which are used through the introduction of dummy variables.

To motivate our approach, we studied 18 explicitly-parallel OpenMP benchmarks from the Rodinia benchmark suite, and found that these benchmarks use six classes of non-affine constructs that are commonly found in parallel scientific applications: 1) Non-affine subscript expressions, 2) Indirect array subscripts, 3) Use of `structs`, 4) Calls to user-defined functions, 5) Non-affine loop bounds, and 6) Non-affine `if` conditions. While there are known techniques from past work to enable automatic analysis for some of these non-affine constructs in polyhedral frameworks, we show that the use of explicit parallelism can enable a larger set of polyhedral transformations for some of these programs (due to conservative dependences), compared to what might have been possible if the input program was sequential.

For future work, we plan to explore how to incorporate additional explicit parallel constructs (such as *barriers*) into dependence polyhedra for increased precision.

9. REFERENCES

[1] The PACE compiler project. *http://pace.rice.edu*.

[2] PLuTo: A polyhedral automatic parallelizer and locality optimizer for multicores. *http://pluto-compiler.sourceforge.net*.

[3] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmotov, C. Nugteren, F. Waters, and A. F. Donaldson. PENCIL: Towards a Platform-Neutral Compute Intermediate Language for DSLs. *CoRR*, abs/1302.5586, 2013.

[4] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 1–10, New York, NY, USA, 2008. ACM.

[5] A. Basumallik and R. Eigenmann. Optimizing Irregular Shared-memory Applications for Distributed-memory Systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 119–128, New York, NY, USA, 2006. ACM.

[6] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Transformations for Communication-minimized Parallelization and

Locality Optimization in the Polyhedral Model. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, pages 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[9] J.-F. Collard and M. Griebl. Array Dataflow Analysis for Explicitly Parallel Programs. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing*, Euro-Par '96, 1996.

[10] R. Cytron. Doacross: Beyond Vectorization for Multiprocessors. In *ICPP'86*, pages 836–844, 1986.

[11] J. Doerfert, C. Hammacher, K. Streit, and S. Hack. SPolly: Speculative Optimizations in the Polyhedral Model. In *Proc. 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, pages 55–61, Berlin, Germany, Jan. 2013.

[12] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *Intl. J. of Parallel Programming*, 21(6):389–420, Dec. 1992.

[13] G. Gan, X. Wang, J. Manzano, and G. R. Gao. Tile Reduction: The First Step towards Tile Aware Parallelization in OpenMP. In *9th International Workshop on OpenMP (IWOMP)*, 2011.

[14] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *Intl. J. of Parallel Programming*, 34(3), 2006.

[15] T. Grosser, A. Größlinger, and C. Lengauer. Polly - Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(4), 2012.

[16] T. Grosser, S. Pop, J. Ramanujam, and P. Sadayappan. Optimistic Delinearization of Parametrically Sized Arrays. In *Proc. 5th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Amsterdam, Netherlands, Jan. 2015.

[17] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. When Polyhedral Transformations Meet SIMD Code Generation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 13)*, Seattle, WA, June 2013. ACM Press.

[18] Y. Lin and D. Padua. Compiler Analysis of Irregular Memory Accesses. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 157–168, New York, NY, USA, 2000. ACM.

[19] V. Maslov and W. Pugh. Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise. Technical report, In CONPAR 94 - VAPP VI, Int. Conf. on Parallel and Vector Processing, 1994.

[20] OpenMP Specifications. http://openmp.org/wp/openmp-specifications.

[21] OpenMP Technical Report 3 on OpenMP 4.0 enhancements. http://openmp.org/TR3.pdf.

[22] W. Pugh and D. Wonnacott. Non-Linear Array Dependence Analysis. In B. Szymanski and B. Sinharoy, editors, *Languages, Compilers and Run-Time Systems for Scalable Computers*, pages 1–14. Springer US, 1996.

[23] J. Shirako, L.-N. Pouchet, and V. Sarkar. Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14, 2014.

[24] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar. Expressing DOACROSS Loop Dependencies in OpenMP. In *9th International Workshop on OpenMP (IWOMP)*, 2011.

[25] A. Simbürger and A. Größlinger. On the Variety of Static Control Parts in Real-World Programs: from Affine via Multi-dimensional to Polynomial and Just-in-Time. Jan. 2014.

[26] M. M. Strout, L. Carter, and J. Ferrante. Compile-time Composition of Run-time Data and Iteration Reorderings. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 91–102, New York, NY, USA, 2003. ACM.

[27] A. Venkat, M. Shantharam, M. Hall, and M. M. Strout. Non-affine Extensions to Polyhedral Code Generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA, 2014. ACM.

[28] D. G. Wonnacott. *Constraint-based Array Dependence Analysis*. PhD thesis, College Park, MD, USA, 1995. UMI Order No. GAX96-22167.

[29] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat. Array Dataflow Analysis for Polyhedral X10 Programs. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, 2013.